Rust for Numerical Applications

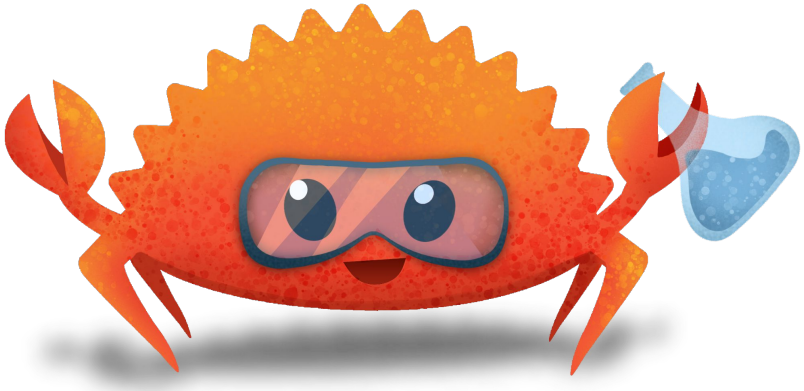
Felipe Zapata

Altas Technologies

Who Am I?

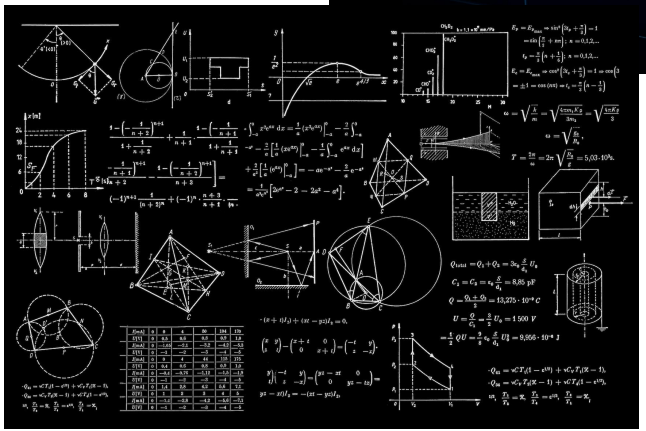
- I’m a software engineer with a background in **scientific simulation**, specifically in physical chemistry
- I develop autonomous trading system using **Rust** and **Python**

Altas Technologies



Need for Speed

- Science
- Finance
- Engineering



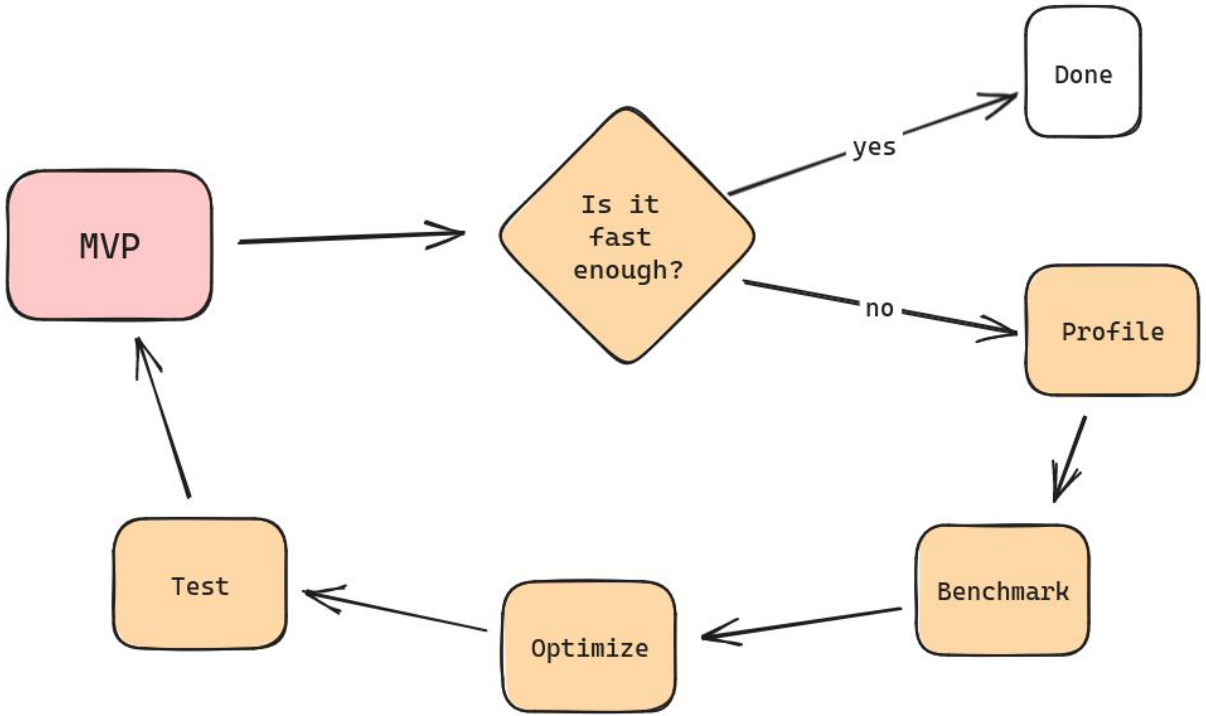
Why Rust for Numerical applications?

- It is blazingly **fast**
- The **tooling** is amazing (bye bye CMake)
- The **borrow-checker** is your friend
- The **ecosystem** got your back covered

The General Algorithm

“I firmly believe that intelligence is just a robust methodology to recursively improve my stupidity”

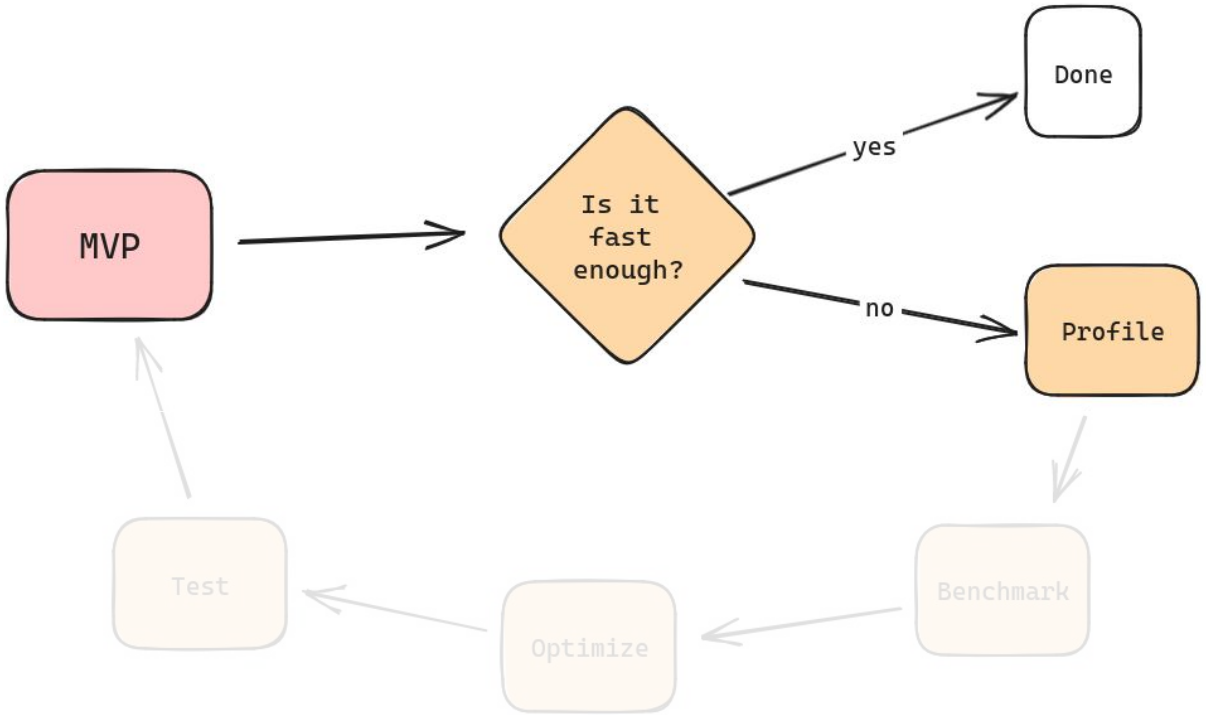
Numerical Applications Recipe



Baby Steps

- Use **Clippy**
- Don’t fight the **borrow checker**
- Use **battle-tested libraries** for performance-critical operations

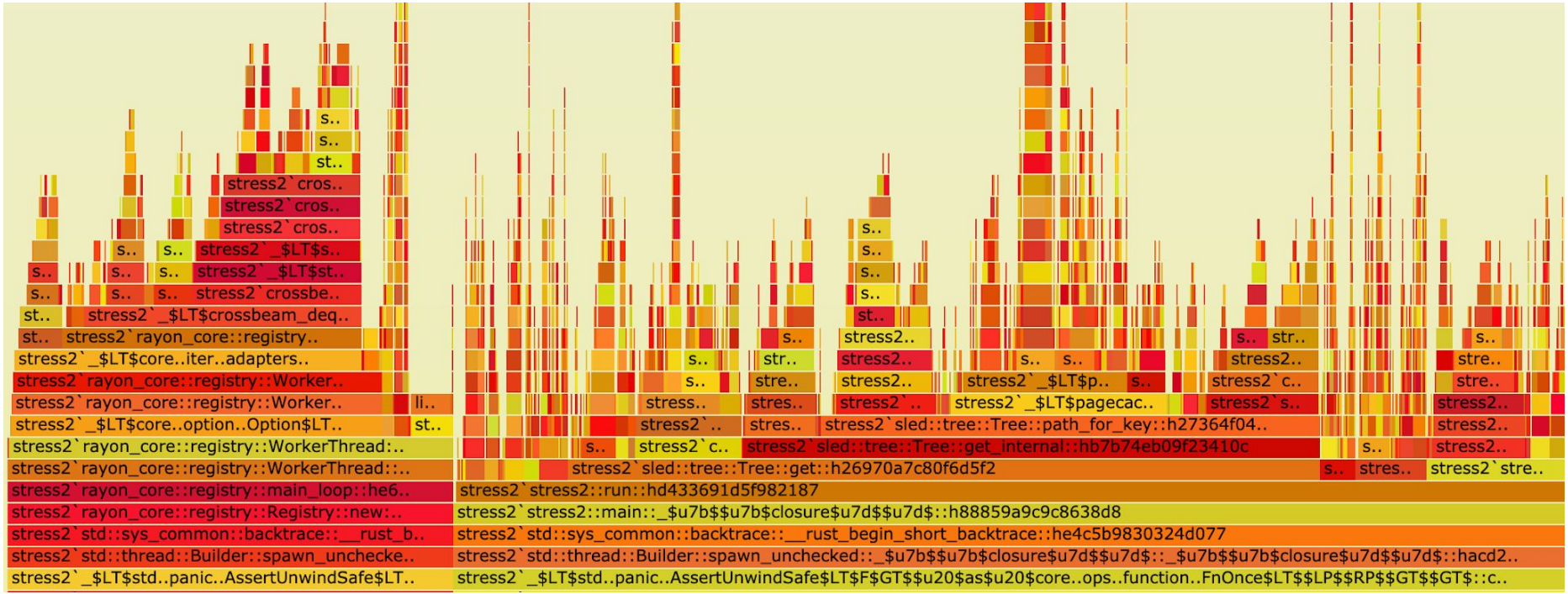
Numerical Applications Recipe



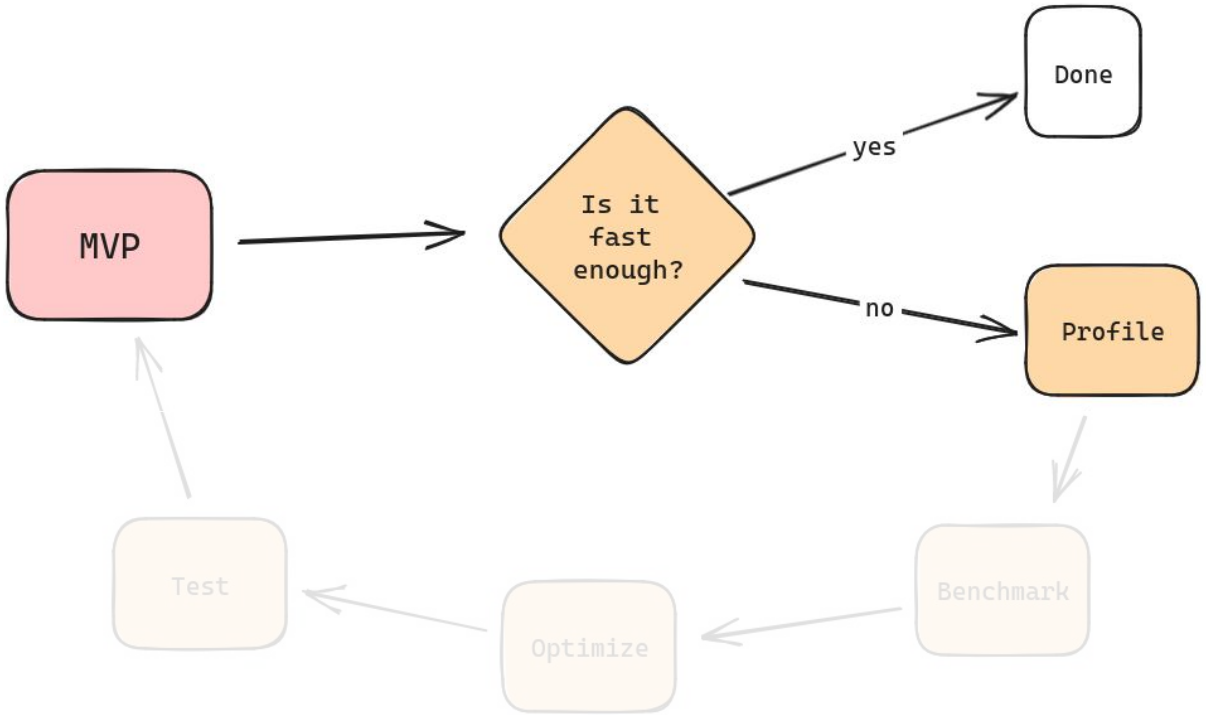
Identify the Bottleneck

- Don’t trust your instincts: **measure**!
- Use a **profiler** tool like Perf
- **Visualize** Perf output with FlameGraphs

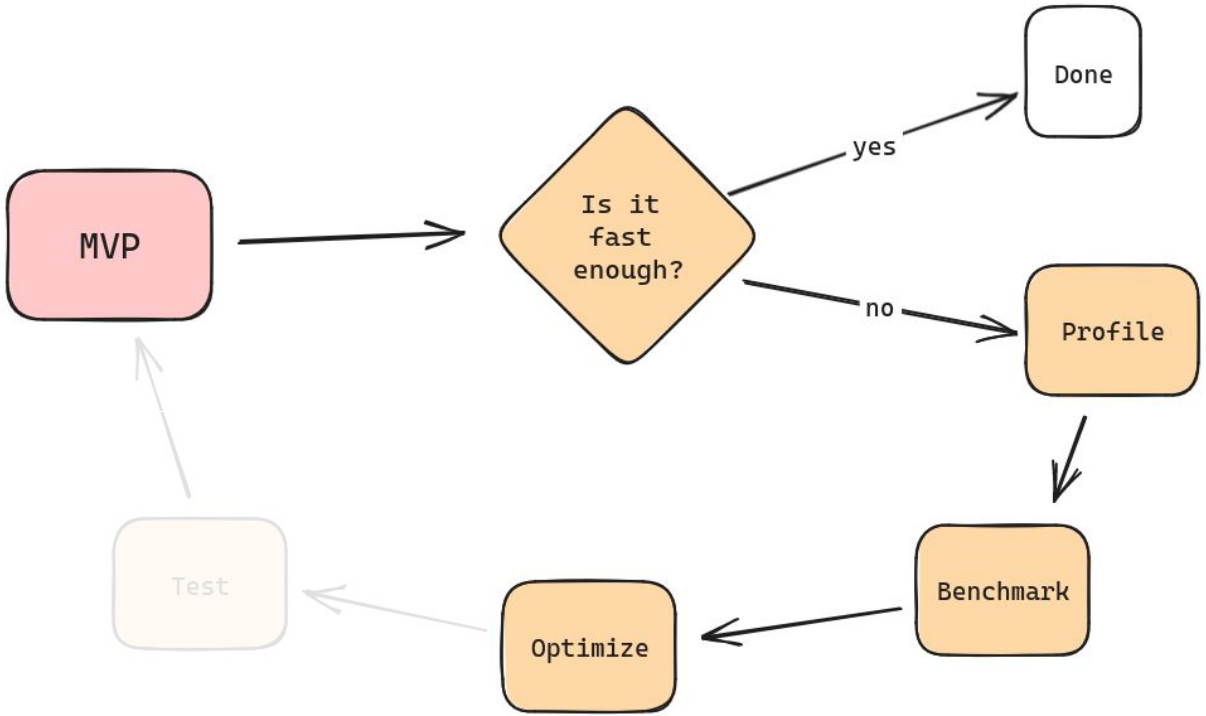
(rust-) Flamegraph



Numerical Applications Recipe



Numerical Applications Recipe



Benchmark: Use criterion

```rust
use criterion::{black_box, criterion_group, criterion_main, Criterion};
use mula::convolve;
use ndarray::{Array, Array1};
use ndarray_rand::rand_distr::Normal;
use ndarray_rand::RandomExt;

pub fn criterion_benchmark(c: &mut Criterion) {
    c.bench_function("convolve 1D", |b| {
        let (arr1, arr2) = generate_arrays::<100>();
        b.iter(|| convolve(black_box(&arr1), black_box(&arr2)))
    });
}

fn generate_arrays<const N: usize>() -> (Array1<f64>, Array1<f64>) {
    let dist = Normal::new(0.0, 1.0).unwrap();
    let arr1 = Array::random([N], dist.clone());
    let arr2 = Array::random([N], dist);
    (arr1, arr2)
}

criterion_group!(benches, criterion_benchmark);
criterion_main!(benches);
```

Optimizations

- Choose the right algorithm
- Do your math homework
- Pre-allocate your vectors: `Vec::with_capacity`
- Use a non-cryptographic hash algorithm for `HashMap`
- Have a look at the Rust perf-book

Benchmarking

First criterion run

```
convolve 1D             time:   [8.4246 µs 8.4905 µs 8.5646 µs]
Found 2 outliers among 100 measurements (2.00%)
  1 (1.00%) high mild
  1 (1.00%) high severe
```

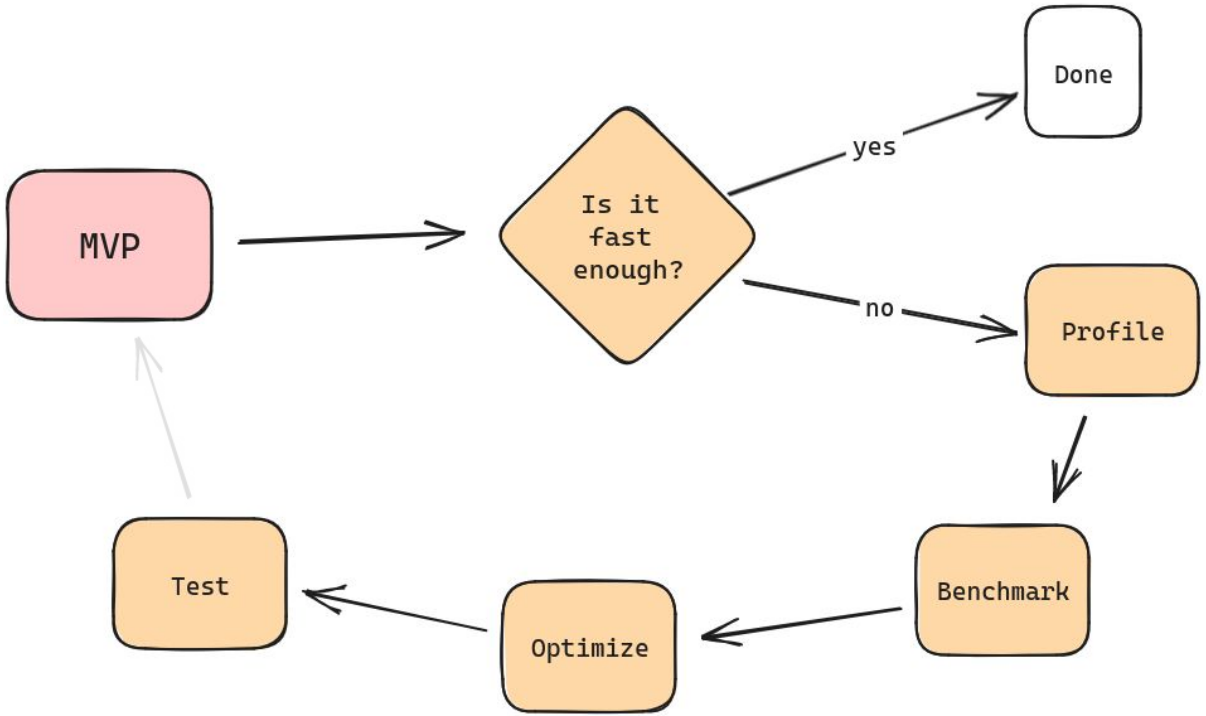
Apply optimization and run again

```
convolve 1D             time:   [8.1579 µs 8.1767 µs 8.1986 µs]
                        change: [-3.0843% -2.4990% -1.9221%] (p = 0.00 < 0.05)
                        Performance has improved.
```

Let's try another optimization

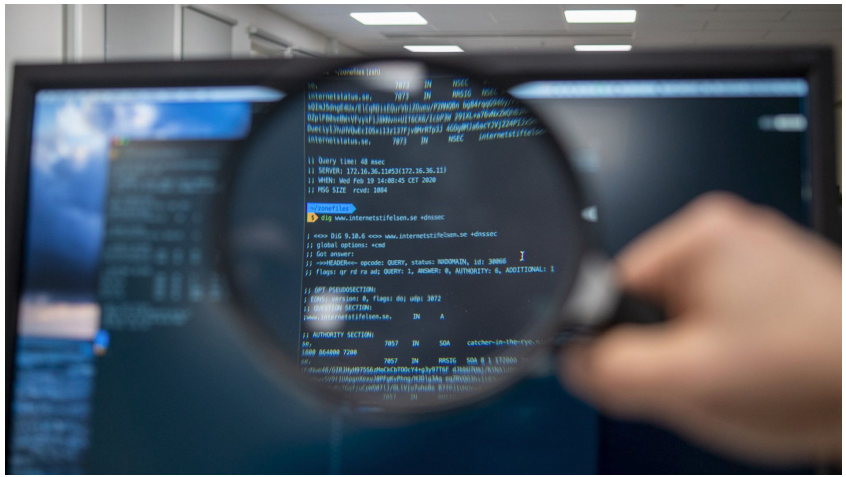
```
convolve 1D             time:   [8.1439 µs 8.1632 µs 8.1848 µs]
                        change: [-0.4452% -0.0142% +0.3841%] (p = 0.95 > 0.05)
                        No change in performance detected.
```

Numerical Applications Recipe



Test

- Ask your favorite LLM to generate **unit tests** for you
- Check **edge cases**
- Use a **property testing framework** like Proptest

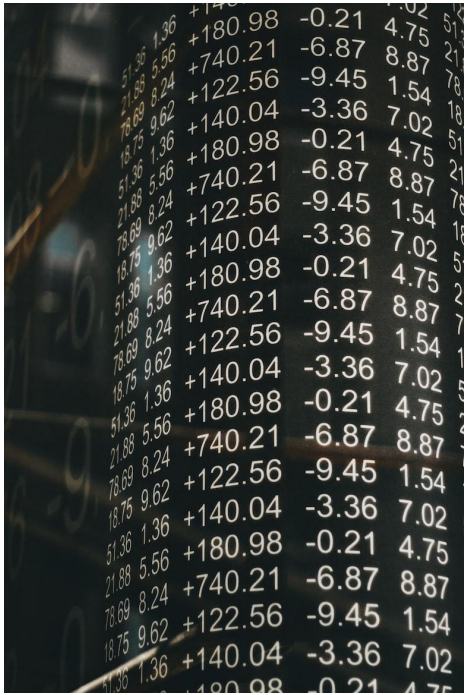


Proptest

```rust
pub fn vec_max(slice: &[f64]) -> f64 {
    slice.iter().fold(f64::NAN, |acc, x| f64::max(acc, *x))
}

pub fn vec_min(slice: &[f64]) -> f64 {
    slice.iter().fold(f64::NAN, |acc, x| f64::min(acc, *x))
}

#[cfg(test)]
mod test {
    use super::{vec_max, vec_min};
    use proptest::prelude::*;

    proptest! {
        #[test]
        fn test_prop_min_max(v in prop::collection::vec(any::<f64>(), 0..1000)) {
            let min = vec_min(&v);
            let max = vec_max(&v);
            prop_assert!(v.iter().all(|x| min <= *x));
            prop_assert!(v.iter().all(|x| max >= *x));
        }
}
```

A Floating Point Errors Footnote

- Floating-point numbers cannot represent all real-numbers accurately (rounding errors)
- Rounding errors can accumulate
- Check rust_decimal for financial calculations



Third-party Libraries

Shall I use an external dependency or shall I cook my own recipe for a given algorithm?

- How central is this algorithm in your calculation?
- How confident are you about implementing the algorithm?
- Are you willing to maintain it?
- What is the quality of the external dependency?

Third-party Libraries

Rule of Thumb:

For other-than-trivial algorithms, use a third-party library - even if it is written in C/C++

Some Pearls for Numerical Applications

- Rust-ndarray family : array manipulation, statistics, linear algebra, etc.
- Rayon: data-parallelism
- Polars: Lighting-fast Dataframe library
- Rustc-hash: A fast non-cryptographic hash algorithm
- Approx: Testing floats approximate equality
- Ordered_float: Wrappers for total order on floats

Rust, Python and Maturin

- Numerical workflows are commonly written in **Python**
- You can easily integrate Rust with Python through Maturin
- Getting familiar with Python numerical ecosystem would greatly benefit your Rust project

Thank You!

Questions?

linkedin.com/in/felipe-zapata

f.zapata@altastechnologies.com